Bellman-Ford

The Bellman–Ford algorithm is a graph shortest-path algorithm that finds the minimum distance from a starting node to all other nodes in a weighted graph. Unlike Dijkstra's algorithm, Bellman–Ford can handle negative edge weights and can also detect negative cycles (loops whose overall weight is negative). It works by repeatedly relaxing all edges—meaning it keeps checking whether a shorter path to a node can be found through another node. This relaxation is done V−1 times (where V is the number of vertices), because the longest possible simple path contains V−1 edges. After these iterations, if any edge can still be relaxed, the graph contains a negative cycle. Bellman–Ford is slower than Dijkstra's (O(V·E) time), but its ability to detect negative cycles makes it very useful in many scenarios.

Advantages

- Handles negative weights: Unlike Dijkstra's algorithm, Bellman–Ford works even when some edges have negative weights.
- Detects negative cycles: It can identify if a graph contains a cycle whose total weight is negative.
- Simple and easy to implement: The logic is straightforward—just relax all edges repeatedly.
- Works for directed and undirected graphs: (For undirected graphs, negative edges must be used carefully to avoid artificial negative cycles.)

Limitations

- Slower than Dijkstra's algorithm: Its time complexity is O(V × E), which is less efficient for large graphs.
- Not suitable for very large networks: Because of high computation time, it becomes heavy when vertices and edges grow.
- Cannot give correct shortest paths if a negative cycle is reachable from the source (though it can detect it).

Graph Representation

Bellman–Ford works on a weighted graph represented as:

- Vertices (nodes)
- Edges with weights (can be positive or negative)

The graph is usually represented as an edge list, because the algorithm needs to check all edges repeatedly.
Example edge representation:

From	To	Weight
A	B	4
B	C	-2
A	C	5

Steps:

Step 1: Initialize Distances

- Choose a source vertex (starting point).
- Create an array dist[] to store shortest distances.
- Set:
    - dist[source] = 0
    - dist[all other vertices] = ∞ (infinity)

This means you initially assume you know nothing except that the distance from the source to itself is zero.

Step 2: Relax All Edges (V − 1 times)

This is the core part.

- A graph with V vertices can have a shortest path with at most V−1 edges.
- So, repeat the relaxation step exactly V−1 times.

Relaxation means:
For every edge (u → v) with weight w:

- If
  dist[u] + w < dist[v],
  then update:
  dist[v] = dist[u] + w

Why do we do this repeatedly?
Because one relaxation may create a new shorter path that helps improve another path in the next iteration.

During these V−1 rounds:

- Each round refines the distances.
- After the final round, all shortest paths will be correctly calculated if no negative cycle exists.

Step 3: Check for Negative-Weight Cycles

After V−1 iterations:

- Do one more relaxation (the V-th time).
- If any distance still gets smaller, it means:

👉 A negative-weight cycle exists.

Because shortest paths would keep decreasing forever in a negative cycle, you cannot compute a valid result in such cases.

Step 4: If No Negative Cycle, Output Results

- If no update happens in Step 3:
    - The algorithm successfully found the shortest paths.
- Output the dist[] array or reconstruct the shortest paths if needed.

📌 Summary (Quick Version)

1. Set all distances to infinity except source = 0.
2. Relax all edges V−1 times.
3. Check for negative cycles by relaxing once more.
4. If no negative cycle, distances are final.

Pseudocode

```
BellmanFord(Graph, source):
  // Step 1: Initialize distances
 for each vertex v in Graph:
```

```
    dist[v] = ∞
dist[source] = 0
// Step 2: Relax edges V - 1 times
for i = 1 to V-1:
    for each edge (u, v, w) in Graph:     // u → v with weight w
        if dist[u] + w < dist[v]:
            dist[v] = dist[u] + w
// Step 3: Check for negative-weight cycle
for each edge (u, v, w) in Graph:
    if dist[u] + w < dist[v]:
        print("Graph contains a negative-weight cycle")
        return
// Step 4: Print distances
print(dist)
```

Time Complexity:

O(V × E)

Where:

- V = number of vertices
- E = number of edges

Why O(V × E)?

- Relaxation of all edges takes O(E) time.
- This is repeated V−1 times → O((V−1) × E) ≈ O(VE).
- The final negative-cycle check takes O(E), but it doesn't change the overall complexity.

Bellman-Ford Code: https://github.com/pulokdas062/Graph-Algorithm/edit/main/Bellmanford/bellmanford.cpp